BOX2D查询手册

Chapter 1 导言

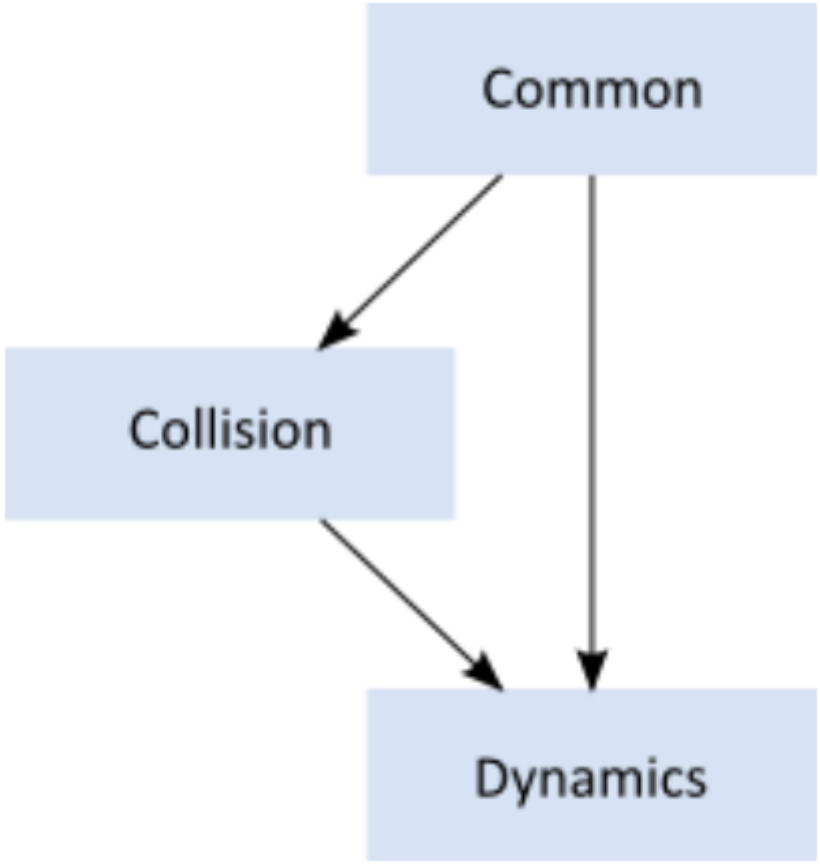
1.1 核心概念

- 形状（shape）、刚体（rigid body）、夹具（fixture）、约束（constraint）、接触约束（contact constraint）、关节（joint）、关节限制（joint limit）、关节马达（joint motor）、世界（world）、求解器（solver）、连续碰撞（continuous collision）

1.2 模块

- Box2D由三个模块组成：通用模块（Common）、碰撞模块（Colision）和力学模块（Dynamics）通用模块

  包含了内存分配、数学和配置的代码。碰撞模块定义形状、碰撞检测和碰撞的函数或队列。最终力学模块提供对世界、物体、夹具和关节的模拟。



1.3 单位

- Box2D使用浮点数，采用米-千克-秒（MKS）单位制

1.4 工厂和定义

表 2.1　b2BodyDef 和 b2FixtureDef 所包含的属性

b2BodyDef 中的属性		
属　性	数据类型	备　注
active	Boolean	刚体可用与否
allowSleep	Boolean	允许睡眠与否
awake	Boolean	活动与否
bullet	Boolean	是否模拟高速子弹，开启 CCD 碰撞检测
angle	Number	刚体角度
angularDamping	Number	刚体旋转时的角速度阻尼
angularVelocity	Number	旋转角速度
fixedRotation	Boolean	是否禁止刚体旋转
inertiaScale	Number	刚体角速度惯性系数
linearDamping	Number	刚体线性速度阻尼
linearVelocity	b2Vec2	刚体线性速度
position	b2Vec2	刚体坐标
type	uint	刚体类型
userData	*	刚体自定义数据
b2FixtureDef 中的属性		
属　性	数据类型	备　注
density	Number	密度
friction	Number	摩擦系数
restitution	Number	弹性系数
filter	b2FilterData	碰撞过滤
isSensor	Boolean	是否为传感器
shape	b2Shape	形状
userData	*	自定义数据

```
//创建函数
b2Body* b2World::CreateBody(const b2BodyDef* def)
b2Joint* b2World::CreateJoint(const b2JointDef* def)
//销毁函数
void b2World::DestroyBody(b2Body* body)
void b2World::DestroyJoint(b2Joint* joint)
//创建夹具：fixture 必须有父 body，所以要使用 b2Body 的工厂方法来创建并销毁它们
b2Fixture* b2Body::CreateFixture(const b2FixtureDef* def)
void b2Body::DestroyFixture(b2Fixture* fixture)
//快速创建夹具
b2Fixture* b2Body::CreateFixture(const b2Shape* shape, float32 density)
```

Chapter 2 Hello Box2D

2.1 创建世界

```
//创建 重力矢量
b2Vec2 gravity(0.0f, -10.0f);
//创建世界
b2World world(gravity);
```

2.2 创建地面盒

body 用以下步骤来创建：

1. 用位置(position), 阻尼(damping)等来定义 body。
2. 用 world 对象来创建 body。
3. 用形状(shape), 摩擦(friction), 密度(density)等来定义 fixture。
4. 在 body 上来创建 fixture。

```
//第一步，创建 ground body
b2BodyDef groundBodyDef;
groundBodyDef.position.Set(0.0f, -10.0f);
//第二步，将 body 定义传给 world 对象，用以创建 ground body。
b2Body* groundBody = world.CreateBody(&groundBodyDef);
//第三步，创建地面多边形-创建一个box盒子形状。SetAsBox 函数接收半个宽度和半个高度作为参数。因此在这种情况下，地面盒就是 100 个单位宽
(x 轴),20 个单位高(y 轴)。
b2PolygonShape groundBox;
groundBox.SetAsBox(50.0f, 10.0f);
```

表 2.2 b2BodyDef 属性与 b2Body 参数函数对应关系

b2BodyDef	b2Body	
属 性	读	写
active	IsActive():Boolean	SetActive(flag:Boolean)
allowSleep	IsSleepingAllowed():Boolean	SetSleepingAllowed(flag:Boolean)
awake	IsAwake():Boolean	SetAwake(flag:Boolean)
bullet	IsBullet():Boolean	SetBullet(flag:Boolean)
angle	GetAngle():Number	SetAngle(angle:Number)
angularDamping	GetAngularDamping():Number	SetAngularDamping(angularDamping:Number)
angularVelocity	GetAngularVelocity():Number	SetAngularVelocity(omega:Number)
fixedRotation	IsFixedRotation():Boolean	SetFixedRotation(fixed:Boolean)
inertiaScale	GetInertia():Number	SetMassData(massData:b2MassData)
linearDamping	GetLinearDamping():Number	SetLinearDamping(linearDamping:Number)
linearVelocity	GetLinearVelocity():b2Vec2	SetLinearVelocity(v:b2Vec2)
position	GetPosition():b2Vec2	SetPosition(position:b2Vec2)
type	GetType():uint	SetType(type:uint)
userData	GetUserData():*	SetUserData(data:*)

表 3.5　b2BodyDef 和 b2FixtureDef 中的属性

类	分　类	属　性	数据类型	备　注
b2BodyDef 中的属性	状　态	active	Boolean	刚体可用与否
		allowSleep	Boolean	允许睡眠与否
		awake	Boolean	活动与否
		bullet	Boolean	是否模拟高速子弹，开启 CCD 碰撞检测
	角度 & 角速度	angle	Number	刚体角度
		angularDamping	Number	刚体旋转时的角速度阻尼
		angularVelocity	Number	旋转角速度
		fixedRotation	Boolean	是否禁止刚体旋转
		inertiaScale	Number	刚体角速度惯性系数
	坐标 & 速度	linearDamping	Number	刚体线性速度阻尼
		linearVelocity	b2Vec2	刚体线性速度
		position	b2Vec2	刚体坐标
	其他属性	type	uint	刚体类型
		userData	*	刚体自定义数据
b2FixtureDef 中的属性	物质特性	density	Number	密度
		friction	Number	摩擦系数
		restitution	Number	弹性系数
	碰撞相关	filter	b2FilterData	碰撞过滤
		isSensor	Boolean	是否为传感器
	形　状	shape	b2Shape	形状
	自定义	userData	*	自定义数据

表 2.3　b2FixtureDef 属性与 b2Fixture 参数函数对应关系

b2FixtureDef	b2Fixture	
属　性	读	写
density	GetDensity():Number	SetDensity(density:Number)
friction	GetFriction():Number	SetFriction(friction:Number)
restitution	GetRestitution():Number	SetRestitution(restitution:Number)
filter	GetFilterData():b2FilterData	SetFilterData(filter:b2FilterData)
isSensor	IsSensor():Boolean	SetSensor(sensor:Boolean)
shape	GetShape():b2Shape	GetShape():b2Shape

表 3.4　b2FixtureDef 属性分类

分　类	属　性	数据类型	备　注
物质特性	density	Number	密度
	friction	Number	摩擦系数
	restitution	Number	弹性系数
碰撞相关	filter	b2FilterData	碰撞过滤
	isSensor	Boolean	是否为传感器
形　状	shape	b2Shape	形状
自定义	userData	*	自定义数据

表 3.1　b2BodyDef 属性分类

分　类	属　性	数据类型	备　注
状　态	active	Boolean	刚体可用与否
	allowSleep	Boolean	允许睡眠与否
	awake	Boolean	活动与否
	bullet	Boolean	是否模拟高速子弹，开启 CCD 碰撞检测
角度 & 角速度	angle	Number	刚体角度
	angularDamping	Number	刚体旋转时的角速度阻尼
	angularVelocity	Number	旋转角速度
	fixedRotation	Boolean	是否禁止刚体旋转
	inertiaScale	Number	刚体角速度惯性系数
坐标 & 速度	linearDamping	Number	刚体线性速度阻尼
	linearVelocity	b2Vec2	刚体线性速度
	position	b2Vec2	刚体坐标
其他属性	type	uint	刚体类型
	userData	*	刚体自定义数据

2.3 创建动态物体

```
//用 CreateBody 创建 body。默认情况下，body 是静态的,所以在构造时候应该设置b2BodyType，使得 body 成为动态的。如果让 body 受力的影响而运动，必须将 body 的类型设为b2_dynamicBody
b2BodyDef bodyDef;
bodyDef.type = b2_dynamicBody;
bodyDef.position.Set(0.0f, 4.0f);
b2Body* body = world.CreateBody(&bodyDef);
//然后，创建多边形 shape，附加到 fixture 定义上
b2PolygonShape dynamicBox;
dynamicBox.SetAsBox(1.0f, 1.0f);

b2FixtureDef fixtureDef;
fixtureDef.shape = &dynamicBox;
//一个动态 body 至少有一个密度不为 0 的 fixture。否则会出现一些奇怪的行为。
fixtureDef.density = 1.0f;
fixtureDef.friction = 0.3f;
```

表 4.1　Box2D 常用刚体操作函数

类	方　法	说　明
b2Body	CreateFixture()	添加 b2Fixture 对象
	CreateFixture2()	添加形状
	DestroyFixture()	删除 b2Fixture 对象
	ApplyForce()	施加作用力
	ApplyImpulse()	施加冲量
	ApplyTorque()	施加旋转扭力
	GetLocalCenter()	获取重心本地坐标
	GetWorldCenter()	获取重心全局坐标
	GetLocalPoint()	将全局坐标点转换为本地坐标
	GetWorldPoint()	将本地坐标点转换为全局坐标
	GetLocalVector()	将全局向量转换成本地向量
	GetWorldVector()	将本地向量转换成全局向量
	GetMass()	获取刚体质量
	SetMassData()	设置刚体质量数据
	Split()	分割刚体
b2Fixture	GetAABB():b2AABB	获取形状最小包围盒子
b2World	QueryAABB()	查找 AABB 碰撞刚体
	QueryShape()	查找形状碰撞刚体
	RayCast()	射线查找

2.4 模拟世界

```
//需要为 Box2D 选取一个时间步,用于游戏的物理引擎需要至少 60Hz 的速度，也就是 1/60 秒的时间步
float32 timeStep = 1.0f / 60.0f;
//约束求解有两个阶段：速度阶段和位置阶段。为每个阶设置迭代次数，次数越高越好，一般>=3即可。
//一次迭代就是在时间步之中的单次遍历所有约束，你可以在单个时间步内多次遍历约束
int32 velocityIterations = 6;
int32 positionIterations = 2;

for (int32 i = 0; i < 60; ++i)
{
 world.Step(timeStep, velocityIterations, positionIterations);
 b2Vec2 position = body->GetPosition();
 float32 angle = body->GetAngle();
 printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle);
}
```

Chapter 3 碰撞模块

3.1 关于

- 碰撞模块包含了形状和操作形状的函数。该模块还包含了动态树(dynamic tree)和 broad-phase，用于加快大型系统的碰撞处理速度。

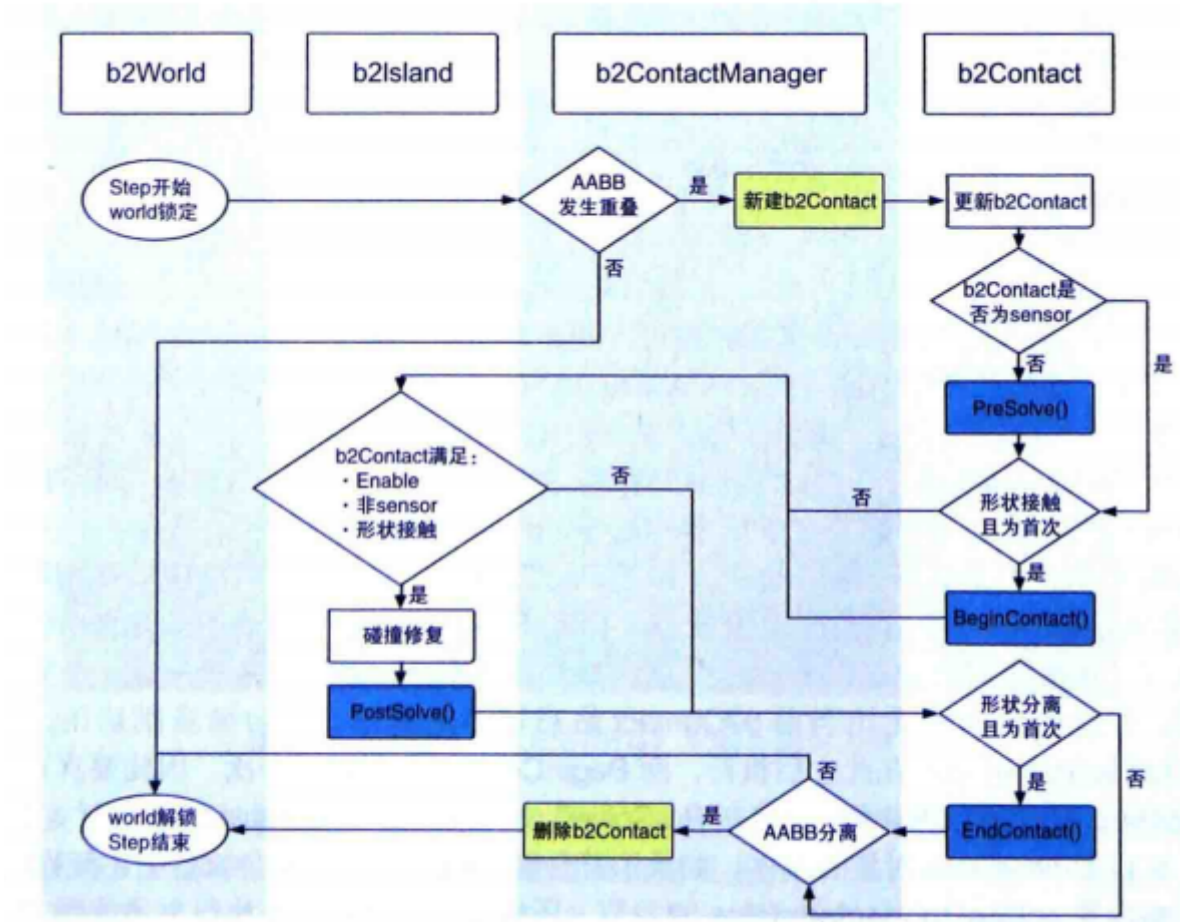


图 5.18　step() 中完整的碰撞运行流程图

表 5.1　b2Contact 类的函数

函　数	说　明
GetFixtureA():b2Fixture	获取发生碰撞的第 1 个 b2Fixture 对象
GetFixtureB():b2Fixture	获取发生碰撞的第 2 个 b2Fixture 对象
GetManifold():b2Manifold	获取碰撞点信息
GetWorldManifold(worldManifold:b2WorldManifold):void	将碰撞点转换成全局坐标，并保存到参数 b2WorldManifold 对象中
IsTouching():Boolean	碰撞双方的形状发生是否接触
IsEnabled():Boolean	是否开启当前的 b2Contact 对象
SetEnabled(flag:Boolean):void	设置当前 b2Contact 对象的状态
IsSensor():Boolean	此次碰撞为是否 sensor
SetSensor(sensor:Boolean):void	设置此次碰撞为 sensor

3.2 形状

b2Shape 是个基类，Box2D 的各种形状都实现了这个基类。此基类定义了几个函数：

- 判断一个点与形状是否有重叠。
- 在形状上执行光线投射(ray cast)。
- 计算形状的 AABB。
- 计算形状的质量。

```
//圆形：圆形有位置和半径。圆形是实心的，你没有办法使圆形变成空心
b2CircleShape circle;
```

```
circle.m_p.Set(2.0f, 3.0f);
circle.m_radius = 0.5f;

//多边形：实心凸多边形，在右手系坐标系下逆时针存储顶点，顶点数最多为8（b2_maxPolygonVertices）
//多边形保护层通过保持多边形的分离来防止隧穿效应。这会导致形状之间有小空隙。你的显示可以比多边形大些，来隐藏这些空隙。
b2Vec2 vertices[3];
vertices[0].Set(0.0f, 0.0f);
vertices[1].Set(1.0f, 0.0f);
vertices[2].Set(0.0f, 1.0f);
int32 count = 3;
b2PolygonShape polygon;
polygon.Set(vertices, count);

//快速创建
void SetAsBox(float32 hx, float32 hy);
void SetAsBox(float32 hx, float32 hy, const b2Vec2& center, float32 angle);

//边框形状：边框形状没有体积，边框形状的主要限制在于它们能够与圆形和多边形碰撞，但它们之间却不会碰撞
//幽灵顶点问题
b2Vec2 v1(0.0f, 0.0f);
b2Vec2 v2(1.0f, 0.0f);
b2EdgeShape edge;
edge.Set(v1, v2);

//链接形状：将许多边框连接在一起，用以构建你的静态游戏世界，自动消除幽灵碰撞，并提供两侧的碰撞。不支持自相交链接形状
b2Vec2 vs[4];
vs[0].Set(1.7f, 0.0f);
vs[1].Set(1.0f, 0.25f);
vs[2].Set(0.0f, 0.0f);
vs[3].Set(-1.7f, 0.4f);
b2ChainShape chain;
chain.CreateChain(vs, 4);

//创建环
b2ChainShape chain;
chain.CreateLoop(vs, 4);
```

3.3 单元几何查询

- 形状点测试

```
//以测试一个点是否与形状有所重叠
b2Transfrom transform;
transform.SetIdentity();
b2Vec2 point(5.0f, 2.0f);
bool hit = shape->TestPoint(transform, point);
```

- 射线检测(Shape Ray Cast)

```
//形状的光线投射，射线在内部则没有交点，链接形状包含儿子索引
```

```
b2Transfrom transform;
transform.SetIdentity();
b2RayCastInput input;
input.p1.Set(0.0f, 0.0f, 0.0f);
input.p2.Set(1.0f, 0.0f, 0.0f);
input.maxFraction = 1.0f;
int32 childIndex = 0;
b2RayCastOutput output;
bool hit = shape->RayCast(&output, input, transform, childIndex);
if (hit)
{
 b2Vec2 hitPoint = input.p1 + output.fraction * (input.p2 - input.p1);
 …
}
```

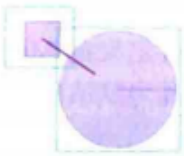
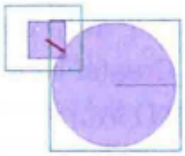
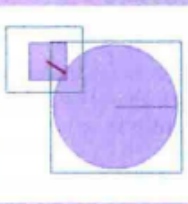
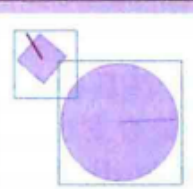
- 对等函数

```
//重叠：测试两个形状是否重叠。
b2Transform xfA = …, xfB = …;
bool overlap = b2TestOverlap(shapeA, indexA, shapeB, indexB, xfA, xfB);

//接触形：圆与圆，圆与多边形的碰撞，得到一个接触点和一个向量；多边形与多边形的碰撞得到两个接触点
b2WorldManifold worldManifold;
worldManifold.Initialize(&manifold, transformA, shapeA.m_radius,
 transformB, shapeB.m_radius);
for (int32 i = 0; i < manifold.pointCount; ++i)
{
 b2Vec2 point = worldManifold.points[i];
 …
}

b2PointState state1[2], state2[2];
b2GetPointStates(state1, state2, &manifold1, &manifold2);
if (state1[0] == b2_removeState)
{
 // process event
}

//距离：b2Distance 函数可以用来计算两个形状之间的距离

//撞击时间（time of impact, TOI）：b2TimeOfImpact 函数用于确定两个形状运动时碰撞的时间。主要目的是防止隧穿效应
```

3.4 碰撞

- 在碰撞发生时，Box2D并没有将碰撞点坐标保存到b2Contact对象中，而是将其保存到了一个叫作b2ManiFold的对象中，我们必须通过b2Contact 的 GetManiFold()，实际使用中用GetWorldManifold()直接获取世界坐标的结果。
- 碰撞侦听器：b2ContactListener

表 5.2　b2ContactListener 碰撞事件处理函数一览表

函数名称	参　数	说　明	调用阶段	示意图
PreSolve	contact: b2Contactold old_Manifold: b2Manifold	只有满足以下条件，才会运行该函数： • Fixture 的 FattenAABB 接触 • Fixture 的 sensor 均为 false • contact 的 isSensor() 返回 false 此时 isTouching() 返回 false 在两个 FattenAABB 的接触过程中 持续调用该函数	潜在碰撞	
BeginContact	contact: b2Contact	此时 isTouching() 返回 true 该函数只在形状首次接触时调用一次 b2Contact 只捕获碰撞的 Fixture， 不记录碰撞点、向量等信息	常规碰撞	
PostSolve	contact: b2Contact impulse: b2ContactImpulse	只有满足以下条件，才会运行该函数： • Fixture 均不是 Sensor • Fixture 形状接触 • contact 的 isSensor() 返回 false • contact 的 isEnabled() 返回 true 此时 isTouching() 返回 true 在两个形状重叠消除之前持续调用 该函数	碰撞修复	
EndContact	contact: b2Contact	此时 isTouching() 返回 false 该函数只在形状首次分离时调用一次 b2Contact 只捕获碰撞的 Fixture， 不记录碰撞点、向量等信息	常规碰撞 分离	

Chapter 4 物体

4.1 物体类型

- staticbody：在模拟时不会运动，物体的质量和质量的倒数存储为零，速度为零，另外也不会和其它static 或 kinematic 物体相互碰撞。
- kinematicbody：在模拟时以一定的速度运动，但不受力的作用。可以让用户手动移动，但通常的做法是设置一定的速度来移动它。Box2D 将它的质量和质量的倒数存储为零。
- dynamicbody：dynamic 物体被完全模拟，受力的作用而运动。dynamic 物体可以和其它所有类型的物体相互碰撞。dynamic 物体的质量总是有限大的，非零的。Box2D 中的物体总是刚体(rigid body)。同一物体上的两个 fixture，永远不会相对移动，也不会碰撞。fixture 有可碰撞的几何形状和密度(density)。物体通常从它的 fixture 中获得质量属性。

4.2 物体定义

- 物体类型

```
//static、kinematic 和 dynami，应该在创建时就确定物体类型，以后再修改代价很高。
bodyDef.type = b2_dynamicBody;
```

- 位置和角度

```
//不要在原点创建物体后再移动它。如果在原点上同时创建了几个物体，性能会很
bodyDef.position.Set(0.0f, 2.0f);
bodyDef.angle = 0.25f * b2_pi;
```

- 阻尼

```
//阻尼用于减小物体在世界中的速度。阻尼跟摩擦有所不同，摩擦仅在物体有接触的时候才会发生。阻尼并不能取代摩擦，往往这两个效果需要同时使用。
bodyDef.linearDamping = 0.0f;
bodyDef.angularDamping = 0.01f;
```

- 重力因子

```
//调整单个物体上的重力
bodyDef.gravityScale = 0.0f;
```

- 休眠参数

```
//当 Box2D 确定一个物体(或一组物体)已停止移动时，物体就会进入休眠状态。休眠物体只消耗很小的 CPU 开销。如果一个醒着的物体接触到了一个休眠中的物体，那么休眠中的物体就会醒过来。当物体上的关节或触点被摧毁的时候，它们同样会醒过来。你也可以手动地唤醒物体。
//手动唤醒
bodyDef.allowSleep = true;
bodyDef.awake = true;
```

- 固定旋转

```
//固定旋转标记使得转动惯量和它的倒数被设置成零。
bodyDef.fixedRotation = true;
```

- 子弹

```
//Box2D 会通过连续碰撞检测(CCD)来防止动态物体穿越静态物体.为了提高行呢，dynamic 物体之间不会应用 CCD。
//高速移动的物体可以标记成子弹(bullet)。子弹跟 static 或者 dynamic 物体之间都会执行 CCD，子弹标记只影响 dynamic 物体。
bodyDef.bullet = true;
```

- 活动状态

```
//不会被其它物体唤醒，其上fixture 也不会 被放到 broad-phase 中。关节可以连接到非活动的物体。但这些关节并不会被模拟。当激活物体时，它的关节不会被扭曲(distorted)
//可以创建一个非活动的物体，之后再激活它
bodyDef.active = true;
```

- 用户数据

```
//用户数据是个 void 指针。它让你将物体和你的应用程序关联起来。你应该保持一致性，所有物体的用户数据都指向相同的对象类型。
b2BodyDef bodyDef;
bodyDef.userData = &myActor;
```

4.3 物体工厂

```
//world 类提供物体工厂来创建和摧毁物体
//Box2D 并不保存物体定义的引用，也不保存其任何数据(除了用户数据指针)。所以你可以创建临时的物体定义，并重复利用它。
b2Body* dynamicBody = myWorld->CreateBody(&bodyDef);
... do stuff ...
myWorld->DestroyBody(dynamicBody);
dynamicBody = NULL;
```

4.4 使用物体

- 质量数据

```
//每个物体都有质量（标量）、质心（二维向量）和转动惯性（标量）
//static 物体质量和转动惯性为零。当物体设置成固定旋转(fixed rotation)，转动惯性也是零。
//改变质量，设置物体的质量后需要reset
void SetMassData(const b2MassData* data);
void ResetMassData();

//得到物体的质量
float32 GetMass() const;
float32 GetInertia() const;
const b2Vec2& GetLocalCenter() const;
void GetMassData(b2MassData* data) const;
```

- 状态信息

```
void SetType(b2BodyType type);
b2BodyType GetType();
void SetBullet(bool flag);
bool IsBullet() const;
void SetSleepingAllowed(bool flag);
bool IsSleepingAllowed() const;
void SetAwake(bool flag);
bool IsAwake() const;
void SetActive(bool flag);
bool IsActive() const;
void SetFixedRotation(bool flag);
bool IsFixedRotation() const;
```

- 位置和速度

```
//位置和旋转角
bool SetTransform(const b2Vec2& position, float32 angle);
const b2Transform& GetTransform() const;
const b2Vec2& GetPosition() const;
float32 GetAngle() const;

//访问本地坐标系及世界坐标下的质心
const b2Vec2& GetWorldCenter() const;
const b2Vec2& GetLocalCenter() const;
```

Chapter 5 夹具

5.1 关于

fixture 具有下列属性：

- 关联的形状
- broad-phase 代理
- 密度(density)、摩擦(friction)和恢复(restitution)
- 碰撞筛选标记(collision filtering flags)
- 指向父物体的指针
- 用户数据
- 传感器标记(sensor flag)

5.2 创建夹具

- 创建夹具

```
//初始化一个 fixture 定义，并将定义传到父物体中
b2FixtureDef fixtureDef;
fixtureDef.shape = &myShape;
fixtureDef.density = 1.0f;
b2Fixture* myFixture = myBody->CreateFixture(&fixtureDef);

//摧毁，物体被摧毁夹具会被自动摧毁
myBody->DestroyFixture(myFixture);
```

- 密度

```
//密度用来计算父物体的质量属性，为零或者是正数。
//所有的 fixture 都应该使用相似的密度，这样做可以改善堆叠稳定性.重设密度需调用 ResetMassData
，使之生效
fixture->SetDensity(5.0f);
body->ResetMassData();
```

- 摩擦

```
//摩擦可以使对象逼真地沿其它对象滑动，支持静摩擦和动摩擦，使用参数相同。摩擦力的强度与正交力（称之为库仑摩擦)成正比。
//摩擦参数通常会设置在 0 到 1 之间，0 意味着没有摩擦，1 会产生强摩擦。
float32 friction;
friction = sqrtf(fixtureA->friction * fixtureB->friction);
```

- 恢复

```
//恢复可以使对象弹起。恢复的值通常设置在 0 到 1 之间
//0 表示非弹性碰撞，1表示完全弹性碰撞
float32 restitution;
restitution = b2Max(fixtureA->restitution, fixtureB->restitution);
```

- 筛选

```
//碰撞筛选是为了防止某些 fixture 之间发生碰撞。Box2D 通过种群和分组来支持筛选
//Box2D 支持 16 个种群。任意 fixture 你都可以指定它属于哪个种群。通过掩码来完成。
playerFixtureDef.filter.categoryBits = 0x0002;
monsterFixtureDef.filter.categoryBits = 0x0004;
playerFixtureDef.filter.maskBits = 0x0004;
monsterFixtureDef.filter.maskBits = 0x0002;

//产生碰撞的规则
uint16 catA = fixtureA.filter.categoryBits;
uint16 maskA = fixtureA.filter.maskBits;
uint16 catB = fixtureB.filter.categoryBits;
uint16 maskB = fixtureB.filter.maskBits;
if ((catA & maskB) != 0 && (catB & maskA) != 0)
{
 // fixtures can collide
}

//碰撞分组让你指定一个整数的组索引。可以让同一个组的所有 fixture 总是相互碰撞(正索引)或永远不碰撞(负索引)
//分组筛选与种群筛选相比，具有更高的优选级。
fixture1Def.filter.groupIndex = 2;
fixture2Def.filter.groupIndex = 2;
fixture3Def.filter.groupIndex = -8;
fixture4Def.filter.groupIndex = -8;

//Box2D 中还有其它的碰撞筛选
//static 上的 fixture 只会与 dynamic 物体上的 fixture 发生碰撞。
//kinematic 物体 只会和 dynamic 物体碰撞。
//同一个物体上的 fixture 永远不会相互碰撞。
//如果两个物体用关节连接起来，物体上面的 fixture 可以选择启用或禁止它们之间相互碰撞。
```

- 传感器

```
//有时候游戏逻辑需要判断两个 fixture 是否相交，而不想有碰撞反应。这可以通过传感器(sensor)来完成。
//传感器可以是 static、kinematic 或 dynamic 的。每个物体上可以有多个 fixture，传感器和实体 fixture 是可以混合存在的。至少一个物体是dynamic 的，传感器才会产生接触事件。 kinematic 与 kinematic 、kinematic 与 static ，或者static 与 static 之间都不会产生接触事件
//获取传感器的两种方法
b2Contact::IsTouching
b2ContactListener::BeginContact 和 EndContact
```

Chapter 6 关节

6.1 关于

- 关节用于把物体约束到世界，或约束到其它物体上。
- 定义：每种关节类型都有各自的定义(definition)，它们都派生自 b2JointDef。所有的关节都连接两个不同的物体。
- 一个关节常常需要一个锚点(anchor point)来定义，这是固定于相接物体中的点。Box2D 要求这些点在局部坐标系中指定
- 关节的生命周期：物体被摧毁，其上关节也会被摧毁

6.2 使用关节

- 创建

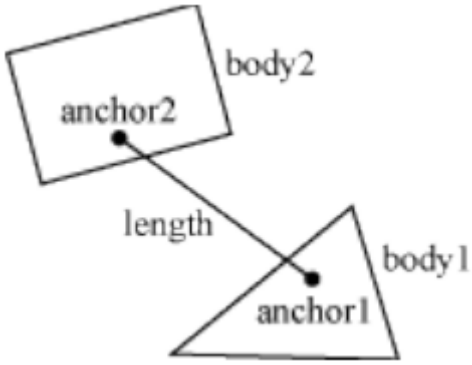
```
b2RevoluteJointDef jointDef;
jointDef.bodyA = myBodyA;
jointDef.bodyB = myBodyB;
jointDef.anchorPoint = myBodyA->GetCenterPosition();
b2RevoluteJoint* joint = (b2RevoluteJoint*)myWorld->CreateJoint(&jointDef);
... do stuff ...
myWorld->DestroyJoint(joint);
joint = NULL;
```

- 使用

```
//在关节上得到物体、锚点和用户数据
b2Body* GetBodyA();
b2Body* GetBodyB();
b2Vec2 GetAnchorA();
b2Vec2 GetAnchorB();
void* GetUserData();

//反作用力和反扭矩
b2Vec2 GetReactionForce();
float32 GetReactionTorque()
```

6.3 距离关节

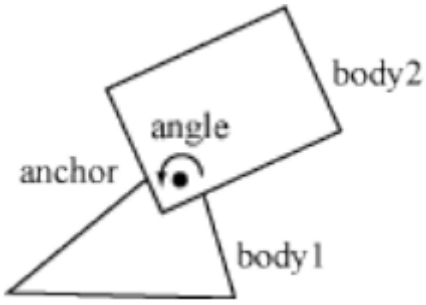
- 两个物体上面各自有一点，两点之间的距离必须固定不变。




```
//定义
b2DistanceJointDef jointDef;
jointDef.Initialize(myBodyA, myBodyB, worldAnchorOnBodyA, worldAnchorOnBodyB);
jointDef.collideConnected = true;

//要使关节有弹性，可以调节一下定义中的两个常数：频率(frequency)和阻尼率(damping ratio)。将频率想象成谐振子(harmonic oscillator，比如吉他弦)振动的快慢。频率使用单位赫兹(Hertz)来指定。典型情况下，关节频率要小于时间步(time step)频率的一半。比如每秒执行 60 次时间步， 距离关节的频率就要小于 30 赫兹。这样做的理由可以参考 Nyquist 频率理论
jointDef.frequencyHz = 4.0f;
jointDef.dampingRatio = 0.5f;
```

6.4 旋转关节

- 旋转关节会强制两个物体共享一个锚点，即所谓铰接点。旋转关节只有一个自由度：两个物体的相对旋转。这称之为关节角。

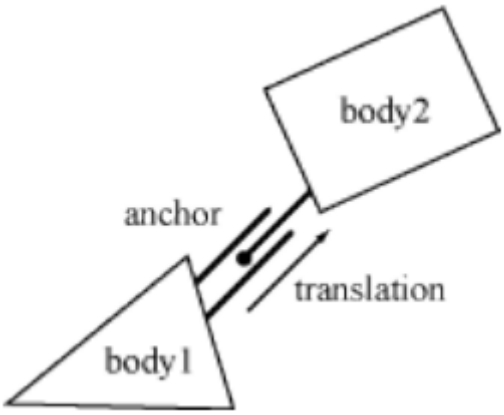



```
b2RevoluteJointDef jointDef;
jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter());

//应该为关节马达提供一个最大扭矩。关节马达会维持在指定的速度。当超出最大扭矩时，关节会慢下来，甚至会反向运动。
b2RevoluteJointDef jointDef;
jointDef.Initialize(bodyA, bodyB, myBodyA->GetWorldCenter());
jointDef.lowerAngle = -0.5f * b2_pi; // -90 degrees
jointDef.upperAngle = 0.25f * b2_pi; // 45 degrees
jointDef.enableLimit = true;
jointDef.maxMotorTorque = 10.0f;
jointDef.motorSpeed = 0.0f;
jointDef.enableMotor = true;
```

```
//访问旋转关节的角度，速度和马达扭矩
float32 GetJointAngle() const;
float32 GetJointSpeed() const;
float32 GetMotorTorque() const;

//每次执行 step 后，也可以更新马达的参数。
void SetMotorSpeed(float32 speed);
void SetMaxMotorTorque(float32 torque);

//用关节马达跟踪想要的关节角
... Game Loop Begin ...
float32 angleError = myJoint->GetJointAngle() - angleTarget;
float32 gain = 0.1f;
myJoint->SetMotorSpeed(-gain * angleError);
... Game Loop End ...
```

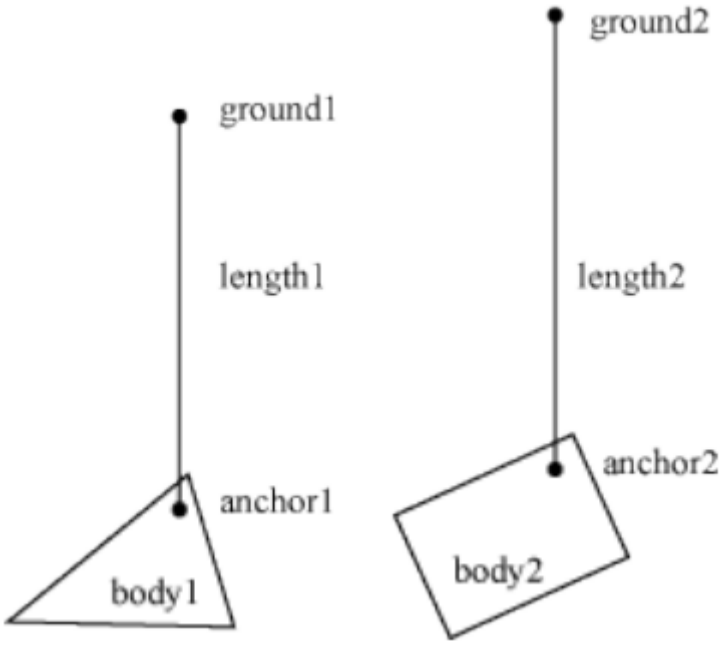
6.5 移动关节

- 移动关节(prismatic joint)允许两个物体沿指定轴相对移动，它会阻止相对旋转。移动关节只有一个自由度。




```
//当使用 Initialize() 创建移动关节时，移动为 0
b2PrismaticJointDef jointDef;
b2Vec2 worldAxis(1.0f, 0.0f);
jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter(), worldAxis);
jointDef.lowerTranslation = -5.0f;
jointDef.upperTranslation = 2.5f;
jointDef.enableLimit = true;
jointDef.maxMotorForce = 1.0f;
jointDef.motorSpeed = 0.0f;
jointDef.enableMotor = true;

//移动关节的用法跟旋转关节类似。相应的成员函数：
float32 GetJointTranslation() const;
float32 GetJointSpeed() const;
float32 GetMotorForce() const;
void SetMotorSpeed(float32 speed);
void SetMotorForce(float32 force);
```

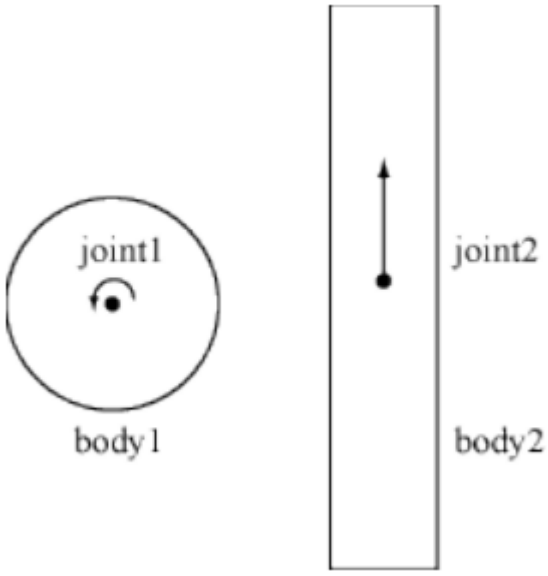
6.6 滑轮关节




```
//滑轮关节用于创建理想的滑轮，它将两个物体接地(ground)并彼此连接。这样，当一个物体上升，另一个物体就会下降。滑轮的绳子长度取决于初始配置。
length1 + length2 == constant
//用系数模拟滑轮组
length1 + ratio * length2 == constant

//定义滑轮
b2Vec2 anchor1 = myBody1->GetWorldCenter();
b2Vec2 anchor2 = myBody2->GetWorldCenter();
b2Vec2 groundAnchor1(p1.x, p1.y + 10.0f);
b2Vec2 groundAnchor2(p2.x, p2.y + 12.0f);
float32 ratio = 1.0f;
b2PulleyJointDef jointDef;
jointDef.Initialize(myBody1, myBody2, groundAnchor1, groundAnchor2, anchor1, anchor2, ratio);
//获取长度
float32 GetLengthA() const;
float32 GetLengthB() const;
```

6.7 齿轮关节

- 齿轮关节只能连接旋转关节和移动关节。




```
//齿轮系数。当一个是旋转关节(有角度的)而另一个是移动关节(平移)时，齿轮系数有长度单位，或者是长度单位的倒数
coordinate1 + ratio * coordinate2 == constant

//定义齿轮
b2GearJointDef jointDef;
jointDef.bodyA = myBodyA;
jointDef.bodyB = myBodyB;
jointDef.joint1 = myRevoluteJoint;
jointDef.joint2 = myPrismaticJoint;
jointDef.ratio = 2.0f * b2_pi / myLength;
```

6.8 其他关节

- 鼠标关节、轮子关节、焊接关节、绳子关节、摩擦关节、马达关节

Chapter 7 接触

7.1 关于

- 接触(contact)是由 Box2D 创建的用于管理 fixture 间碰撞的对象。如果 fixture 有诸如链接形状之类的子 fixture ，那么每个相应的子 fixture 都存在接触。
- 接触点：接触点就是两个形状相互接触的点。
- 接触发现：接触法线是一个单位向量，由一个 fixture 指向另一个 fixture。按照惯例，向量由 fixtureA 指向fixtureB。
- 接触分隔：分隔正好与穿透(penetration)相反。当形状相重叠时，分隔为负。
- 接触流形：两个凸多边形相互接触，有可能会产生两个接触点。这些点都有相同的法线，所以它们被分成一组，构成接触流形，这是连续区域接触的一个近似。
- 法向冲量：法向力作用于接触点，用于防止形状相互穿透。为方便起见，Box2D 使用冲量(impulses)。法向力

  与时间步相乘，构成法向冲量。
- 切向冲量：切向力会在接触点生成，用于模拟摩擦。为方便起见，切向作用使用冲量的方式存储。

- 接触标识：Box2D 试图复用上一个时间步计算出的接触力，做为下一个时间步的初始估计值。Box2D 使用接

  触标识匹配跨越时间步的触点。

7.2 接触类

- 访问接触流形

```
b2Manifold* GetManifold();
const b2Manifold* GetManifold() const;

//修改，但不提倡
void GetWorldManifold(b2WorldManifold* worldManifold) const;

//传感器
bool touching = sensorContact->IsTouching();

//非传感器，从接触(contact)中你可以得到 fixture，从而再得到 body。
b2Fixture* fixtureA = myContact->GetFixtureA();
b2Body* bodyA = fixtureA->GetBody();
MyActor* actorA = (MyActor*)bodyA->GetUserData()
```

7.3 访问接触

```
//在 world 中，可以遍历所有的接触：
for (b2Contact* c = myWorld->GetContactList(); c; c = c->GetNext())
{
 // process c
}

//在 body 中，也可以遍历所有接触。接触以图的方式存储，使用了接触边数据结构(contact edge
structure)
for (b2ContactEdge* ce = myBody->GetContactList(); ce; ce = ce->next)
{
 b2Contact* c = ce->contact;
 // process c
}
```

7.4 接触监听器

```
//接触监听器支持几种事件：开始(begin)，结束(end)，求解前(pre-solve)和求解后(post-solve)。
class MyContactListener : public b2ContactListener
{
public:
 void BeginContact(b2Contact* contact)
 { /* handle begin event */ }
 void EndContact(b2Contact* contact)
 { /* handle end event */ }
 void PreSolve(b2Contact* contact, const b2Manifold* oldManifold)
 { /* handle pre-solve event */ }
 void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)
 { /* handle post-solve event */ }
};
```

- 开始接触事件：当两个 fixture 开始有重叠时，事件会被触发。传感器和非传感器都会触发这事件。这事件只能在

  时间步内发生。
- 接触接触事件：当两个 fixture 不再重叠时，事件会被触发。传感器和非传感器都会触发这事件。当一个 body 被

  摧毁时，事件也有可能被触发。所以这事件也有可能发生在时间步之外。
- 求解前事件：在碰撞检测之后，但在碰撞求解之前，事件会被触发。这样可以给你一个机会，根据当前的配置来决定是否使这个接触失效。

```
//举例：在回调中使用 b2Contact::SetEnabled(false)，可以实现单向门（允许一侧的物体无障碍的穿过，而另一侧的物体无法穿过）的功能
void PreSolve(b2Contact* contact, const b2Manifold* oldManifold)
{
 b2WorldManifold worldManifold;
 contact->GetWorldManifold(&worldManifold);
 if (worldManifold.normal.y < -0.5f)
 {
 contact->SetEnabled(false);
 }
}

//如果要确认触点状态或得到碰撞前的速度，可以在 pre-solve 事件中处理
void PreSolve(b2Contact* contact, const b2Manifold* oldManifold)
{
 b2WorldManifold worldManifold;
 contact->GetWorldManifold(&worldManifold);
 b2PointState state1[2], state2[2];
 b2GetPointStates(state1, state2, oldManifold, contact->GetManifold());
 if (state2[0] == b2_addState)
 {
 const b2Body* bodyA = contact->GetFixtureA()->GetBody();
 const b2Body* bodyB = contact->GetFixtureB()->GetBody();
 b2Vec2 point = worldManifold.points[0];
 b2Vec2 vA = bodyA->GetLinearVelocityFromWorldPoint(point);
 b2Vec2 vB = bodyB->GetLinearVelocityFromWorldPoint(point);
 float32 approachVelocity = b2Dot(vB - vA, worldManifold.normal);
 if (approachVelocity > 1.0f)
 {
```

```
 MyPlayCollisionSound();
 }
 }
}
```

- 求解后事件

```
//示例：在操作接触缓冲时，如何处理孤立物体
// 我们打算摧毁和 contact 指针有关联的物体。
// 我们必须先缓存那些需要摧毁的物体，因为它们有可能被多个触点所共有。
const int32 k_maxNuke = 6;
b2Body* nuke[k_maxNuke];
int32 nukeCount = 0;
// 遍历 contact 缓存，摧毁那些正在和更重的物体接触的物体。
for (int32 i = 0; i < m_pointCount; ++i)
{
 ContactPoint* point = m_points + i;
 b2Body* bodyA = point->fixtureA->GetBody();
 b2Body* bodyB = point->FixtureB->GetBody();
 float32 massA = bodyA->GetMass();
 float32 massB = bodyB->GetMass();
 if (massA > 0.0f && massB > 0.0f)
 {
 if (massB > massA)
 {
 nuke[nukeCount++] = bodyA;
 }
 else
 {
 nuke[nukeCount++] = bodyB;
 }
 if (nukeCount == k_maxNuke)
 {
 break;
 }
 }
}
// 将 nuke 数组排序，使得重复的指针归在一起
std::sort(nuke, nuke + nukeCount);
// 删除 body，忽略重复的
int32 i = 0;
while (i < nukeCount)
{
 b2Body* b = nuke[i++];
 while (i < nukeCount && nuke[i] == b)
 {
 ++i;
 }
 m_world->DestroyBody(b);
}
```

7.5 接触筛选

- 通过实现 b2ContactFilter 类，Box2D 允许定制接触筛选。

```
bool b2ContactFilter::ShouldCollide(b2Fixture* fixtureA, b2Fixture* fixtureB)
{
const b2Filter& filterA = fixtureA->GetFilterData();
const b2Filter& filterB = fixtureB->GetFilterData();
if (filterA.groupIndex == filterB.groupIndex && filterA.groupIndex != 0)
{
return filterA.groupIndex > 0;
}
bool collide = (filterA.maskBits & filterB.categoryBits) != 0 &&
 (filterA.categoryBits & filterB.maskBits) != 0;
return collide;
}

//在运行期（run-time），你可以创建自己的接触筛选实例，并使用 b2World::SetContactFilter 函数来注册。 要保证当 world 存在时，你filter 要保留在作用域中
MyContactFilter filter;
world->SetContactFilter(&filter);
// filter 留在作用域中
```

Chapter 8 世界类

8.1 使用

- 创建和摧毁

```
b2World* myWorld = new b2World(gravity, doSleep);
... do stuff ...
delete myWorld;
```

- 模拟

```
float32 timeStep = 1.0f / 60.f;
int32 velocityIterations = 10;
int32 positionIterations = 8;
myWorld->Step(timeStep, velocityIterations, positionIterations);
```

- 探测：可以获取世界中所有物体、接触和关节并遍历它们

```
//唤醒世界中的所有物体
for (b2Body* b = myWorld->GetBodyList(); b; b = b->GetNext())
{
 b->SetAwake(true);
}
```

- AABB查询：有时你需要得出一个区域内的所有 fixture。b2World 类为此使用了 broad-phase 数据结构，并提供

  了一个 log(N) 的快速方法

```
class MyQueryCallback : public b2QueryCallback
{
public:
 bool ReportFixture(b2Fixture* fixture)
 {
 b2Body* body = fixture->GetBody();
 body->SetAwake(true);

 // Return true to continue the query.
 return true;
 }
};
...
MyQueryCallback callback;
b2AABB aabb;
aabb.lowerBound.Set(-1.0f, -1.0f);
aabb.upperBound.Set(1.0f, 1.0f);
myWorld->Query(&callback, aabb)
```

- 光线投射

```
// This class captures the closest hit shape.
class MyRayCastCallback : public b2RayCastCallback
{
public:
 MyRayCastCallback()
 {
 m_fixture = NULL;
 }

 float32 ReportFixture(b2Fixture* fixture, const b2Vec2& point,
const b2Vec2& normal, float32 fraction)
 {
 m_fixture = fixture;
 m_point = point;
 m_normal = normal;
 m_fraction = fraction;
 return fraction;
 }

 b2Fixture* m_fixture;
 b2Vec2 m_point;
 b2Vec2 m_normal;
 float32 m_fraction;
};
MyRayCastCallback callback;
b2Vec2 point1(-1.0f, 0.0f);
b2Vec2 point2(3.0f, 1.0f);
myWorld->RayCast(&callback, point1, point2);
```

- 力与冲量：可以将力、扭矩和冲量应用到物体上。当应用一个力或者冲量时，你需要提供一个在世界坐标下的受力点。这经常导致相对于质心，会有个扭矩。

```
void ApplyForce(const b2Vec2& force, const b2Vec2& point);
void ApplyTorque(float32 torque);
void ApplyLinearImpulse(const b2Vec2& impulse, const b2Vec2& point);
void ApplyAngularImpulse(float32 impulse);
```

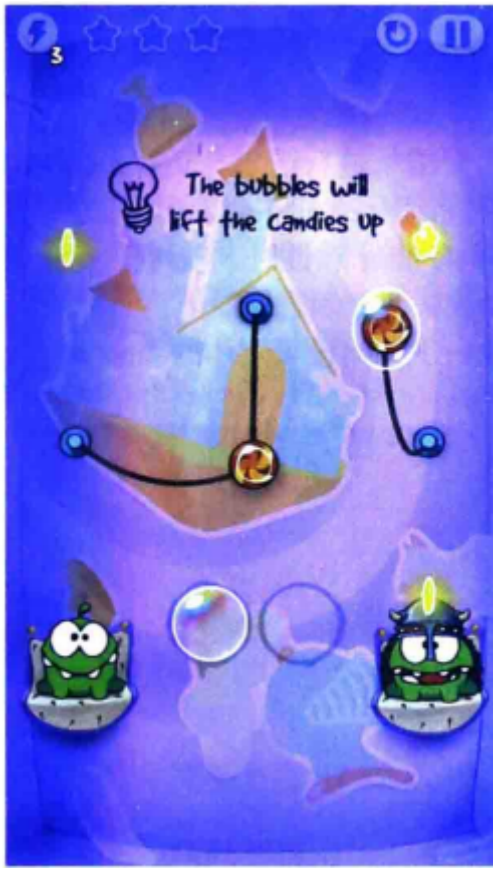
- 坐标转换

```
//内联时会更高效
b2Vec2 GetWorldPoint(const b2Vec2& localPoint);
b2Vec2 GetWorldVector(const b2Vec2& localVector);
b2Vec2 GetLocalPoint(const b2Vec2& worldPoint);
b2Vec2 GetLocalVector(const b2Vec2& worldVector);
```

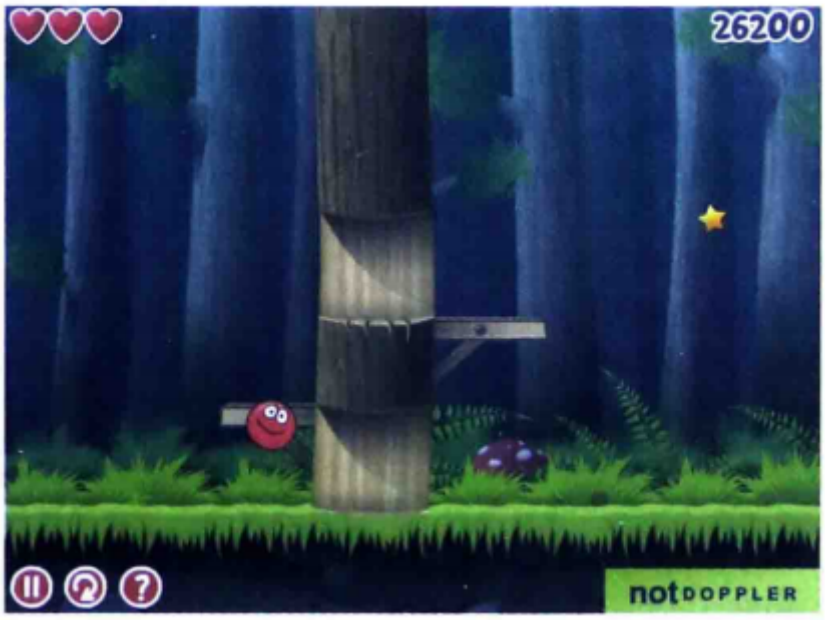
Chapter 9 物理效果

9.1 游戏案例内效果

- 《割绳子》效果、气泡效果（sensor检测）

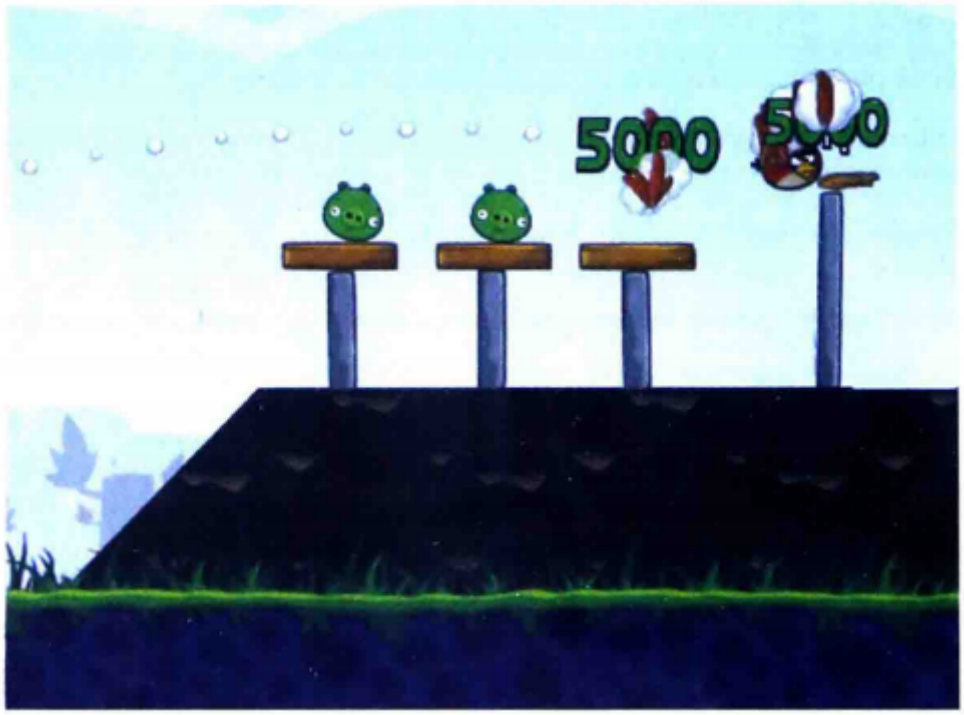


- 《红球历险记》单向板（contact检测，P132，P151）



- 《愤怒小鸟》定向万有引力（P142）

- 《愤怒小鸟》冲撞速度伤害，速度越快伤害越高（冲量，P147）



- 《超越重力》碰撞黏贴，碰撞到物体后黏贴在表面（P163）

9.2 其他效果

- 体积膨胀、物体碎裂（Split分离多个物体）

Chapter 10 杂项

10.1 用户数据

```
//举个典型的例子，角色上附有物体，并在物体中附加角色的指针，这就构成了一个循环引用。如果你有角色(actor)，你就能得到物体。如果你有物体，你也能得到角色。
GameActor* actor = GameCreateActor();
b2BodyDef bodyDef;
bodyDef.userData = actor;
actor->body = box2Dworld->CreateBody(&bodyDef);
```

10.2 限制

Box2D 使用了一些数值近似来让模拟更高效。这就带来一些限制。

这是当前的限制：

1. 将重的物体放到相对很轻的物体上面，会不稳定。当质量比到 10:1 时，稳定性就会降低。
2. 用关节将 body 链接起来，如果是较轻的物体吊着较重的物体，body 链接有可能被拉伸。比如，一条很轻的锁链吊着个很重的球，就可能不稳定。当质量比超过 10:1 时，稳定性就会降低。
3. 通常还有约 0.5cm 的间隙，就检测到形状与形状碰撞。
4. 连续碰撞不会处理关节，因此你会看到在快速移动的物体上的关节拉伸。
5. Box2D 使用欧拉积分法，它不会导致物体的抛物线运动，并只有一阶导数的精度。然而这种方法足够快并有很好的稳定性。
6. Box2D 使用迭代求解器来提供实时计算。你不可能得到绝对准确的刚体碰撞和像素。增加迭代的次数会提升准确性。